CSC 263, Summer 2018, Midterm

Student ID No.	UTorID
Last Name	First Name
Email	Signature

- **Questions 1–10 are multiple choice questions, 2 points each. Choose one option for each question. No need to justify your answer. You must fill in your answers to the table below to get marks.**
- **Answer the other questions in the spaces given in the questions.**
- **For Questions 16–20, answer briefly when you describe your algorithm, justify its correctness, or analyze the running time. You may use procedures, algorithms, proofs, and other results we gave in lectures/tutorials/assignments directly without explaining the details.**
- **Present your student ID on the front, right corner of your desk. One-page letter-size reference sheet is allowed. It is not allowed to use phones, electronic watches, or any other electronic devices.**

Fill in your answers to Questions 1–10 below:

1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	6	7	8	9	10
D	C	C	C	D	C	D	C	A	C

Questions	Marks
1 – 10 (20 points)	
11 – 16 (40 points)	
17 – 20 (50 points)	
Total (110 points)	

1. (*2 points*) Which one of the following is false?

- A. $O(n) \subset O(n \log n)$
- B. $O(n \log n) \subset O(n^2)$
- C. $O(n^2) \subset O(n!)$
- D. $O(n!) \subset O(2^n)$

2. (*2 points*) Which one of the following is false?

- A. $\Omega(n) \supset \Omega(n^2)$
- B. $\Omega(n^2) \supset \Omega(2^n)$
- C. $\Omega(n^2) \subset O(2^n)$
- D. $\big(\Omega(n) \cap O(n^2)\big) \supset \big(\Omega(n^2) \cap O(n)\big)$

3. (*2 points*) Suppose we need a data structure storing a collection of elements, and we require it to support the following operations efficiently:

- Search: search by a key value.
- ExtractMin: find the minimum element and remove it from the collection.
- ExtractMax: find the maximum element and remove it from the collection.

Which of the following is the best choice?

- A. max-heap
- B. min-heap
- C. AVL tree
- D. hash table with chains

4. (*2 points*) Which one of the following is wrong?

- A. The max-heap property is that every node is no less than its children.
- B. The max-heap property is that every node is no less than its descendants.
- C. The binary search tree property is that every node is no less than its left child and no greater than its right child.
- D. The binary search tree property is that every node is no less than all nodes in its left subtree and no greater than all nodes in its right subtree.

5. (*2 points*) In the array representation for storing an n-element heap, the number of leaves is:

- A. $\lfloor \log n \rfloor$
- B. $\lceil \log n \rceil$
- C. $\lfloor n/2 \rfloor$
- D. $\lceil n/2 \rceil$

6. (*2 points*) For AVL trees, which one of the following is possible as the tree size (expressed in term of the tree height h)? Recall that the size of a tree is the number of nodes in the tree.

- A. $\Theta(h^{2.1})$
- B. $\Theta(1.1^h)$
- C. $\Theta(1.9^h)$
- D. $\Theta(2.1^h)$

7. (*2 points*) Consider an improved implementation of hash tables with chaining, where the chains are implemented using AVL trees instead of linked lists. What is the best asymptotic bound for the running time of searching in such a hash table, under the simple uniform hashing assumption? Let α be the load factor of the hash table.

- A. $O(1+\alpha)$
- B. $O((1+\alpha)/(\log \alpha))$
- C. $O(\log \alpha)$
- D. $O(\max\{1, \log \alpha\})$

8. (*2 points*) Consider the following procedure computing a dummy function. Which one is a correct asymptotic bound for the running time of F(N) expressed in terms of N?

```
F($i$):
if $i < 3$ then
    return 0
end if
return F($i-1$) + F($i-2$)
```

- A. $O(N)$
- B. $O(N^2)$
- C. $O(2^N)$
- D. $O(N \log(N))$

9. (*2 points*) Consider another procedure computing a dummy function. What is the best asymptotic bound for the running time of F'(N), expressed in terms of N?

```
F'($i$):
$a = b = 0$
for $j = 3$ to $i$ do
    $c = a + b$
    $a = b$
    $b = c$
end for
return b
```

- A. $O(N)$
- B. $O(N^2)$
- C. $O(2^N)$
- D. $O(N \log(N))$

10. (*2 points*) We consider a Monte Carlo algorithm for finding an element larger than the median in an unsorted array, where the median is given. The algorithm runs by simply picking an element randomly and returning it. An improvement of this algorithm is to repeat it k times and find the maximum of the k returned values. What is the probability that this improvement gives a wrong answer? Suppose the input array has n distinct elements and n is even; the median is the $\frac{n}{2}$-th element when the array is sorted.

- A. $1/2$
- B. $1/(2k)$
- C. $1/2^k$
- D. k/n

11. (*5 points*) **Ternary max-heap.** In a ternary max-heap, every node has exactly three children (except for the leaves and, possibly, one node with fewer than three children). The values stored in each node are ordered according to the same principle as for binary heaps: the value at each node is greater than or equal to the values in its children. Just like for binary heaps, we would like to store the heap elements in an array (without creating a linked structure).

We use the first element storing the root, that is, the index of the root is 1. Given the index i of an element, write an exact expression for each of the following:

(a) (1 point) The index of the left child of i: ________________ $3i - 1$

(b) (1 point) The index of the middle child of i: ________________ $3i$

(c) (1 point) The index of the right child of i: ________________ $3i + 1$

(d) (2 points) The index of the parent of i: ________________ $\lfloor \frac{i+1}{3} \rfloor$ or $\lceil \frac{i-1}{3} \rceil$

12. (*5 points*) **Max-heap operations.** Consider the following max-heap in its array representation.

13	10	8	5	2

(a) (2 points) Give the resulting max-heap after inserting 15.

15	10	13	5	2	8

(b) (3 points) Give the resulting max-heap after extracting the maximum from the result in (a).

13	10	8	5	2

13. (*5 points*) **Insertion for double hashing.** Consider an open-addressing hash table T where its hash function is defined as

$$\begin{aligned} h(k,i) &= (h_1(k) + ih_2(k)) \mod 10, \\ h_1(k) &= k \mod 10, \\ h_2(k) &= 1 + (k \mod 7) \end{aligned}$$

Fill in the slots below by inserting the following sequence of keys into an empty T: 12, 103, 9, 5, 35, 13, 22, 16. (The first three keys 12, 103, 9 are already inserted.)

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
		12	103						9

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
13		12	103	22	5	35		16	9

14. (*5 points*) **AVL tree insertion and rebalancing.** Draw the final AVL tree structure after inserting the following sequence of keys (in the given order) into an empty tree:
10, 15, 13, 5, 2, 8.

```
        10
       /   \
      5     13
     / \      \
    2   8      15
```

15. (*15 points*) **Comparing data structures.** For each of the data structures in the following table (except hash table), what is the best asymptotic worst-case running time for each operation listed? Fill in using $O(\cdots)$ in terms of the number of elements n. (Answers for *sorted array* are given as examples.) For hash table with chains, give the average-case running time under the simple uniform hashing assumption, and assume the chains are implemented using doubly linked lists and the load factor is a constant.

For max-heap, the parameter i is the index of an element in its array representation. For other structures, the parameter x is an object reference, containing necessary links (previous/next attributes for linked lists and parent/children attributes for trees).

For Insert(A, x), assume $x.key$ does not exist in the structure; for Delete(A, x) and Successor(A, x), assume x is already stored in the structure.

You do not need to justify your answer. Every wrong answer deducts 1 point, until no points left for this question.

	Search(A, k)	Insert(A, k)	Delete(A, i)	Successor(A, i)	Maximum(A)
Sorted array	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Max-heap	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(1)$

	Search(A, k)	Insert(A, x)	Delete(A, x)	Successor(A, x)	Maximum(A)
Sorted, doubly linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Unsorted, doubly linked list	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
AVL tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Hash-table with chains	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$

16. (*5 points*) **Decrease key in max-heaps.** Write the pseudocode to implement DecreaseKey(A, i, k) in max-heaps, that is, given a max-heap $A[1, \ldots, n]$, decrease $A[i]$ to a smaller value k. Your algorithm must run in time $O(\log n)$. Justify that your algorithm is correct and runs in $O(\log n)$.

```
DecreaseKey($A, i, k$):
if $A[i] < k$ then
    error "new key is larger"
end if
$A[i] = k$
MaxHeapify($A, i$)
```

MaxHeapify will bubble down the new smaller key to the right position. Since MaxHeapify runs in $O(\log n)$, so is DecreaseKey.

17. (*15 points*) **Largest k numbers in sorted order.** Given a set of n numbers, we wish to find the k largest in sorted order using a comparison-based algorithm. Consider the following algorithms, implemented using the most efficient way as we see in class. Analyze the running times of each algorithm: give the best asymptotic worst-case running time for (a)–(d) and expected running time for (e), using $O(\cdot)$ in terms of both n and k. Briefly justify your answer.

(a) Sort the numbers using merge sort, and list the k largest.

(b) Build a max-heap from the numbers, and call ExtractMax k times.

(c) Build an AVL tree, call Maximum to find the largest, and then call Predecessor k times.

(d) Use randomized quick select algorithm to find the kth largest number, partition around that number, and sort the k largest numbers.

(e) Analyze the expected running time for the algorithm in (d).

We first note that $k \leq n$.

(a) $O(n \log n)$.

Merge sort takes $O(n \log n)$. List the k largest in the sorted result takes $O(k)$.

(b) $O(n + k \log n)$.

Building max-heap takes $O(n)$. Each call of ExtractMax takes $O(\log n)$; k calls takes $O(k \log n)$.

(c) $O(n \log n)$.

Building AVL tree takes $O(n \log n)$. Maximum takes $O(\log n)$, and k calls of Precessor takes $O(k \log n)$.

(d) $O(n^2)$.

Quick select in the worst case runs in $O(n^2)$. Sorting k number of takes $O(k \log k)$.

(e) $O(n + k \log k)$.

Randomized quick select runs in expected $O(n)$. Sorting k number of takes $O(k \log k)$.

It depends on the value k to compare which one is the best. If we consider expected running time, the best one is (e). For worst-case running time, the best is (b).

18. (*10 points*) **Efficient k-way merging.** Consider the problem of merging k sorted lists into one sorted list. Let n_i be the number of elements in the i-th list, for $i = 1, \ldots, k$. Let $n = \sum_{i=1}^{k} n_i$ be the total number of elements in all the input lists.

(a) (2 points) The first algorithm we consider is to iteratively merge the lists; that is, merge the first two lists, and then merge the result with the 3rd list, and so on until all lists are merged. For merging two sorted lists, we can use the MERGE method of MERGESORT. What is the running time of this algorithm? Express using $O(\cdot)$ in terms of n and k.

(b) (2 points) The second algorithm is to merge all k lists together. That is, we continuously find the smallest element among the k input lists, and move it to the output. Since the first element in each list is the smallest in that list, we can compare the first elements in all lists and find the minimum in $O(k)$. What is the running time of this algorithm?

(c) (6 points) Give an $O(n \log k)$-time algorithm to solve this problem. You may just describe your algorithm in English without writing the pseudocode:

- specify the data structures used in your algorithm, including elements stored in the structures and operations needed;
- clearly describe the idea of your algorithm, listing its major steps;
- justify that the running time is $O(n \log k)$.

(a) $O(nk)$.

The first merge takes time $O(n_1 + n_2)$. The second one takes time $O(n_1 + n_2 + n_3)$. ... The last ($k-1$-th) one takes time $O(n_1 + n_2 + \ldots + n_k) = O(n)$. The total running time is $O((k-1)n_1 + (k-2)n_2 + \ldots + 2n_{k-1} + n_k) = O(nk)$.

(b) $O(nk)$.

Finding the smallest among k elements takes $O(k)$. We need run n times of this process of finding the smallest, the total time is $O(nk)$.

(c) There are two ways to do so, either one is correct.

The first one generalizes (a). We only use arrays as the data structures. Instead of merging lists one by one, we merge the lists by pairs. In the first round, pair up the n lists, and merge each pair of two lists; this generates $\lceil n/2 \rceil$ lists. In the second round, continue this processing merging of pairs, and generate $\lceil n/4 \rceil$ lists. In $O(\log n)$ rounds, we will have one list left, which is the result. Each round takes $O(n)$ since we only need to process each element once in each round. Therefore the total time is $O(n \log n)$.

The second approach generalizes (b). We use min-heap as the data structure. Each element of the heap contains an element and also the index of the list that contains this element. We need EXTRACTMIN to extract the minimum from the heap, and INSERT to add an element to the heap.

We first build up a min-heap of size k, by moving the minimum element from each of the k lists; this takes $O(k)$. Then, we continuously extract the minimum from the heap and put it into the output; suppose this minimum is from the i-th list, we also insert the minimum of the remaining elements in the i-th list. Both extraction and insertion takes $O(\log k)$. We need to do this n times to move all n elements. Thus it takes $O(n \log k)$ time in total.

19. (*15 points*) **AVL tree augmentation** This question asks you to augment AVL tree to efficiently implement the following operations in $O(\log n)$:

- CountGEQ$(root, k)$: Return the number of elements with a key greater than or equal to a given value k.
- CountRange$(root, a, b)$: Return the number of elements with a key in the range $[a, b)$, that is, the number of elements x such that $a \le x.key < b$.

(a) (4 points) What kinds of additional information (attributes) will you store at each node of the AVL tree? Explain how to maintain this new information during Insert and Delete, without affecting the $O(\log n)$ running time bound.

(b) (7 points) Write down the pseudocode to implement CountGEQ. Briefly explain your algorithm, and then justify the running time.

(c) (4 points) Write down the pseudocode to implement CountRange. Briefly explain your algorithm, and then justify the running time.

(a) size.

For each node x, we need an attribute $x.size$ which stores the number of nodes in the subtree rooted at x. In the class we have shown it can be maintained efficiently.

(It can be updated by the following rule: $x.size = x.left.size + x.right.size + 1$. Insertion and deletion will only affect the sizes of the nodes along the path from the inserted/deleted node up to the root. Since the height is $O(\log n)$, we can updates the affected nodes in $O(\log n)$.)

(b)
```
CountGEQ($root, k$):
if $root == NIL$ then
    return 0
end if
if $k > root.key$ then
    return CountGEQ($root.right, k$)
end if
return CountGEQ($root.left, k$) $+ root.right.size + 1$ // $k \le root.key$
```

We recursively go down the tree from the root. If k is larger than the key of the current node, it means all nodes in the left subtree have keys $< k$; we discard the left subtree and go down to the right subtree. Otherwise, all nodes in the right subtree have keys $\ge k$; we add up the right subtree size and 1 (the current node), and go down to the left subtree. Since the tree height is $O(\log n)$, we recursively call CountGEQ $O(\log n)$ times. The total running time is $O(\log n)$.

(c)
```
CountRange($root, a, b$):
return CountGEQ($root, a$) - CountGEQ($root, b$)
```

Since CountGEQ is in $O(\log n)$, the total running time is $O(\log n)$.

20. (*10 points*) **Majority element.** An array $A[1, \ldots, n]$ (elements are in arbitrary order) is said to have a *majority element* if more than half of its elements are the same. This question asks you to design an efficient algorithm to tell whether there exists a majority element in a given array, and, if so, to find that element. Describe your algorithm and the data structures you use, justify that it is correct, and analyze its running time. Do not need to write the pseudocode, but clearly list the main steps of your algorithm.

Your marks will be based on the correctness and efficiency of your algorithm. You will get full marks if your algorithm is correct and runs in linear time in the worst case, and get less marks if your algorithm is less efficient (0 mark if your algorithm is wrong). (*Don't spend too much time on this question.*)

There are many ways to solve this problem. We give eight algorithms below.

Algorithms (1)–(2) are in worst-case time $O(n)$. They both use the idea of majority voting, that is, to pair up elements and eliminate pairs with different elements. [These two solution can get 10 points maximum.]

Algorithm (3) is a randomized algorithm, runs in $O(n)$ to get error probability ϵ, for any constant ϵ, and in time $O(nk)$ to get error probability $1/2^k$. [This solution can get 7 points maximum.]

Algorithm (4) uses hash table to count the frequency of each element. It runs in $O(n)$ on average, under the simple uniform assumption. [This solution can get 7 points maximum.]

Algorithm (5) uses the divide-and-conquer technique. It runs in $O(n \log n)$ in the worst case. [This solution can get 5 points maximum.]

Algorithm (1)–(5) do not require comparing elements, which is a classical requirement for this problem. (The elements may not be comparable, such as images.)

Algorithm (6)–(7) uses sorting or AVL tree. They run in $O(n \log n)$ in the worst case, and do require comparing elements. [This solution can get 5 points maximum.]

Algorithm (8) is a trivial brute-force nested loop, running in $O(n^2)$. [This solution can get 3 points maximum.]

We need a subroutine CheckMajority(A, b) to check whether or not b is a majority element of $A[1, \ldots, n]$. It will just count the number of times b appears in A and compare the count with $n/2$; this runs in time $O(n)$.

I use the same name FindMajority for all the solutions below, but you should consider these solutions are independent, and only need to give one of them.

(1) The first algorithm recursively reduce the size the input array by at least half. We pair up elements in A, getting $\lfloor n/2 \rfloor$ pairs. (For the single element left when n is odd, we call CheckMajority; return it if it is a majority, and discard it otherwise.) For each pair of elements, if they are different, discard both of them; if they are the same, keep one copy. After one round of pairing up and elimination, we have at most $n/2$ elements left. We continue this until no element left. The pseudocode is as follows.

```
FindMajority($A$):
Let $n$ be the number of elements in $A$
if $n$ is odd then
    if CheckMajority($A, A[n]$) then
        return $A[n]$
    end if
end if
Initialize an empty array $B$
```

```
for $i = 1$ to $n/2$ do
    if $A[2i-1] == A[2i]$ then
        Add $A[2i]$ to $B$
    end if
end for
if $B$ not empty then
    $b =$ FindMajority($B$)
    if $b \neq$ "no majority" and CheckMajority($A, b$) then
        return $b$
    end if
end if
return "no majority"
```

Th correctness of this algorithm is by the argument that, if there is a majority b in A, then b must remains as a majority in B. (Note this holds only in one direction; this is why we need call CheckMajority for the returned value of FindMajority(B) to verify if the majority of B is actually a majority in A.)

Suppose there is a majority b in A. Let $\#b$ be the number of times b appears, and let $|A| - \#b$ be the number of times any other element appears; then $\#b > |A| - \#b$.

First, after discarding any element $c \neq b$, we still have b as a majority since $\#b$ does not change but $|A|$ reduces by 1. So, we only need to consider when A is even.

We can think of discarding pairs with distinct elements one after another. If we discard a pair with no b, then b is still a majority since $\#b$ does not change but $|A|$ reduces by 2. If we discard a pair containing (one) b, then b remains a majority because $\#b > |A| - \#b$ still holds (both $\#b$ and $|A| - \#b$ reduce by 1). Thus, after discarding all pairs with two different elements, we still have b as a majority.

Since the remaining pairs are all with two same elements, keeping one for each pair does not change the majority. Therefore, we showed that, if b is a majority in A, then b must also be a majority in B; furthermore, the size of B is at most half of the size of A.

Since each recursive call of FindMajority reduces the input array size by half, and the call itself takes linear time (two calls of CheckMajority), the total running time is $O(n) + O(n/2) + O(n/4) + \ldots = O(n)$.

(2) This algorithm is similar to the previous one. But instead of eliminating elements by pairing up, we keep a count for a candidate majority, and do pairing up and elimination implicitly by updating the count.

```
FindMajority($A$):
Let $n$ be the number of elements in $A$
$count = 0$ // count for the candidate
for $i = 1$ to $n$ do
    if $count == 0$ then
        $candidate = A[i]$ // update candidate
        $count = 1$
    else if $A[i] == b$ then
        $count++$
    else
        $count--$
        end if
    end for
    if $count > 0$ and CheckMajority($A, candidate$) then
        return $candidate$
    end if
    return "no majority"
```

The argument of the correctness of this algorithm is similar to the previous algorithm. Suppose A has a majority b. Whenever the count go down by 1, we are essentially eliminating a pair of two different elements. As argued before, no matter whether or not the pair contains b, b remains majority in the rest of the elements. Since b appears $> n/2$ times, at the end, the candidate must be it and $count > 0$.

We need call CheckMajority for the candidate to verify if it is the majority. (A counter example is [1, 2, 3, 2, 1], which has no majority.)

The algorithm obviously runs in $O(n)$ since it goes through each element once.

(3) Now we consider a randomized algorithm FindMajority. That is, we randomly pick an element b from A, and then call CheckMajority to verify if it is a majority. If there is a majority b, then with probability at least $\lceil \frac{n}{2} \rceil / n > 1/2$ we give the correct answer; the error probability is $p = \lfloor \frac{n}{2} \rfloor / n < 1/2$. If there is no majority, we will return "no majority" for sure, due to the call of CheckMajority.

The error probability can be reduced by repetition. If we repeat k times, the error probability can be reduced to $(1-p)^k < 1/2^k$. For $k = O(\log n)$, we can reduce the error to $1/n$.

Running one time takes $O(n)$, due to CheckMajority. Running k times takes $O(nk)$.

You might think of running the algorithm without CheckMajority. *That is, randomly pick an element and return it. But this way gives large error probability. If there is a majority, then the error probability is $p = \lfloor \frac{n}{2} \rfloor / n < 1/2$, the same as before. But if there is no majority, then the error probability is 1 (always wrong).*

A straightforward improvement is to repeat k times to generate k samples, for an odd k, and then find and return the majority in the k samples in $O(k)$ using one of the linear time algorithm shown before.

If there is a majority b, then we get wrong answers when b appears $< k/2$ times in the samples. Let $p = \lfloor \frac{n}{2} \rfloor / n < 1/2$. The error probability is $\sum_{i=0}^{\lfloor k/2 \rfloor} p^{k-i}(1-p)^i < 1/2$ (but this probability is actually still close to $1/2$, not reduced much; you have to repeat very large number of times).

If there is no majority, the worst-case scenario is when n is even and there are exactly two different elements, each appearing $n/2$ times. This time the error probability is still 1, since there is no majority in the original array but we will always find one in k samples since k is odd.

You can also think of choosing k to be even when n is even, and k to be odd when n is odd. This gives error probability $1 - \binom{n}{n/2}\frac{1}{2^k} < 1$. Though this is < 1, again it needs too many repetition to bring it down.

(4) This algorithm is to use a hash table with chaining, and the load factor is a constant. We use the element value as the key, and also maintain a count for each key. By going through

each element a in the array A, we first search a in the hash table: if it is in the hash table, update its count by adding 1; if not, insert it and initialize the count to be 1.

At the end, go through each element again, and find the count by searching in the hash table. If any element has count $> n/2$, we return it; otherwise, return "no majority".

Under the simple uniform hashing assumption, searching and insertion of each element takes $O(1)$ on average. Since we go through the array two times, the total time is $O(n)$ on average.

(5) This algorithm uses the divide-and-conquer technique (similar to MergeSort). We break down the input array by half, and recursively find the majorities in the left half and in the right half. If the two returned majorities are the same, return it; otherwise, call CheckMajority for both to find out which might be the majority (it may be that neither is actually a majority).

```
FindMajority($A$):
Let $n$ be the number of elements in $A$
**if** $n == 1$ **then**
    **return** $A[1]$
**end if**
Let $B = A[1, \dots, \lfloor n/2 \rfloor]$ // left half
Let $C = A[\lfloor n/2 \rfloor + 1, \dots, n]$ // right half
$left$ = FindMajority($B$)
$right$ = FindMajority($C$)
**if** $left == right$ **then**
    **return** $left$
**else if** $left \neq$ "no majority" and CheckMajority($A, left$) **then**
    **return** $left$
**else if** $right \neq$ "no majority" and CheckMajority($A, right$) **then**
    **return** $right$
**else**
    **return** "no majority"
**end if**
```

The correctness of this algorithm follows from the property that if A has a majority b, then b must be either a majority in the left half or a majority in the right half. We prove this below.

Let n_1, n_2 be the numbers of elements in the left half and the right half. Let $\#b_1, \#b_2$ be the numbers of b's in the left half and the right half. Then $n = n_1 + n_2$ and $\#b = \#b_1 + \#b_2$. Since $\#b > n/2$, that is, $\#b_1 + \#b_2 > n_1/2 + n_2/2$, we get at least one of the following inequalities holds: $\#b_1 > n_1/2$, $\#b_2 > n_2/2$. This means b is either a majority in the left half, or a majority in the right half, or both.

Since each recursive call divides the array by half, we have the recursion tree at most $O(\log n)$ levels. The running time at each level takes $O(n)$ (due to CheckMajority). Therefore, the total running time is $O(n \log n)$.

(6) We can simply sort the input array using MergeSort, and then go through each element in the sorted result to count the frequency. If any element has frequency $> n/2$, return it; otherwise, return "no majority". The running time is $O(n \log n)$.

(7) We can also use a balanced binary search tree (such as AVL tree) to count the frequency

(similar as in the use of hash table). That is, each node of the tree contains the element value as the key, and also maintains its count. We go through each element of the array A, we first search a in the tree: if it is in the tree, update its count by adding 1; if not, insert it and initialize the count to be 1. At the end, go through each node of tree. If any element has count $> n/2$, we return it; otherwise, return "no majority".

Searching and insertion in AVL tree takes $O(\log n)$. Since we go through the array two times, the total time is $O(n \log n)$.

(8) A trivial algorithm is to use a nested loop to count. Use the outer loop to go through each element, and use the inner loop to count how many time that element appears. The running time is $O(n^2)$.